Instruction Set Architectures

Part I: From C to MIPS

Readings: 2.1- 2.14

Goals for this Class

- Understand how CPUs run programs
  - How do we express the computation the CPU?
  - How does the CPU execute it?
  - How does the CPU support other system components (e.g., the OS)?
  - What techniques and technologies are involved and how do they work?
- Understand why CPU performance (and other metrics) varies
  - How does CPU design impact performance?
  - What trade-offs are involved in designing a CPU?
  - How can we meaningfully measure and compare computer systems?
- Understand why program performance varies
  - How do program characteristics affect performance?
  - How can we improve a programs performance by considering the CPU running it?
  - How do other system components impact program performance?

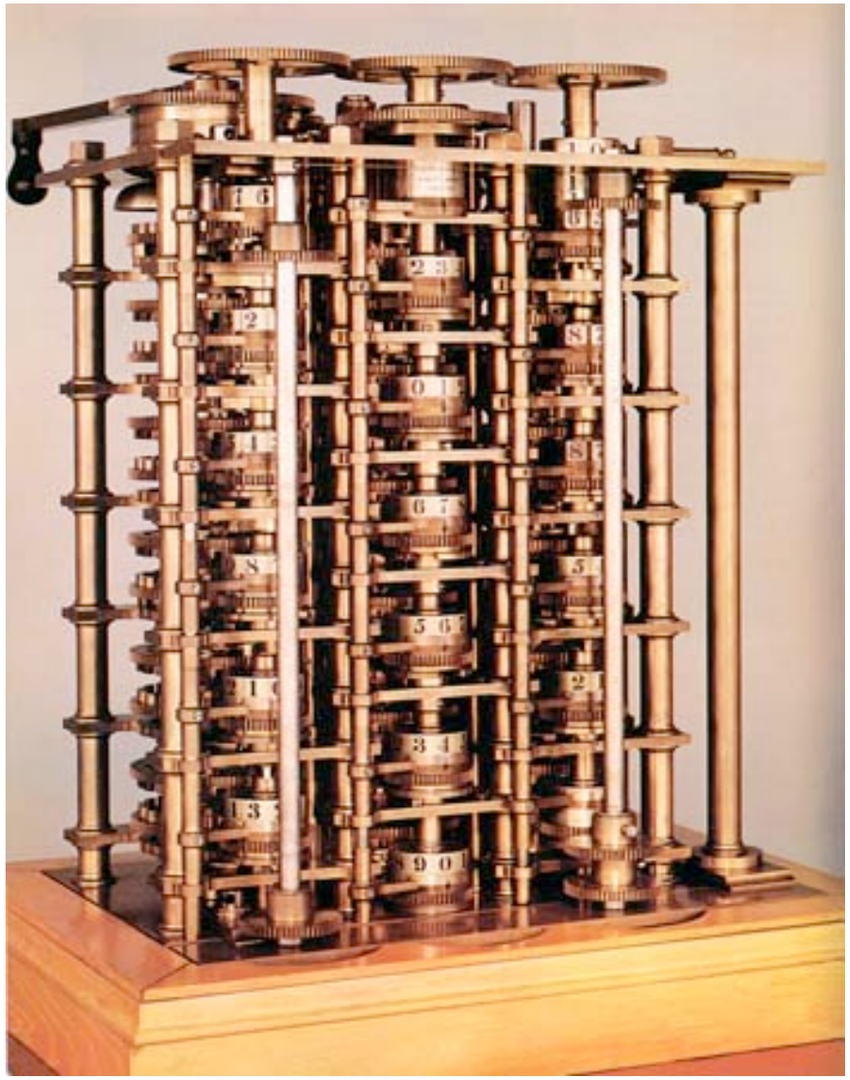
Goals

- Understand how we express programs to the computer.
  - The stored-program model
  - The instruction set architecture
- Learn to read and write MIPS assembly
- Prepare for your 141L Project and 141 homeworks
  - Your book (and my slides) use MIPS throughout
  - You will implement a subset of MIPS in 141L
- Learn to “see past your code” to the ISA
  - Be able to look at a piece of C code and know what kinds of instructions it will produce.
  - Begin to understand the compiler’s role
  - Be able to roughly estimate the performance of code based on this understanding (we will refine this skill throughout the quarter.)

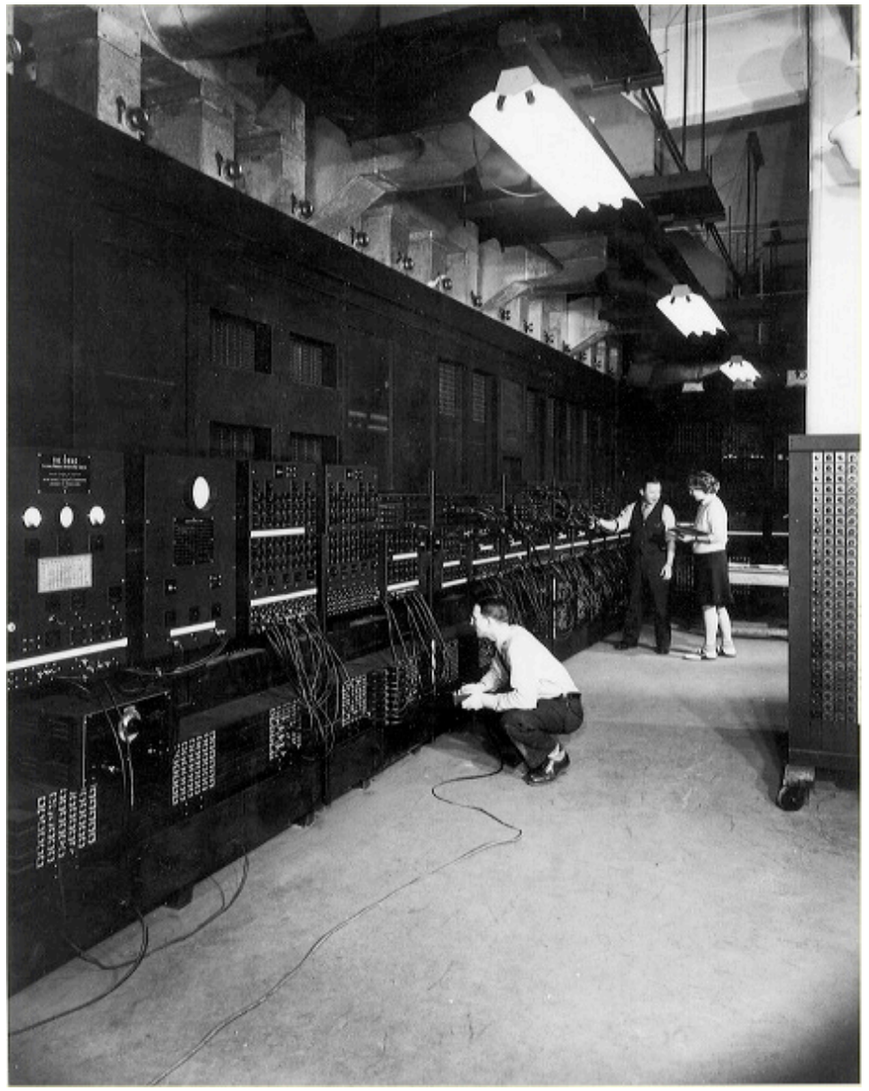
The Idea of the CPU

In the beginning...

- Physical configuration specified the computation a computer performed



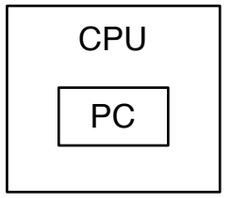
The Difference Engine

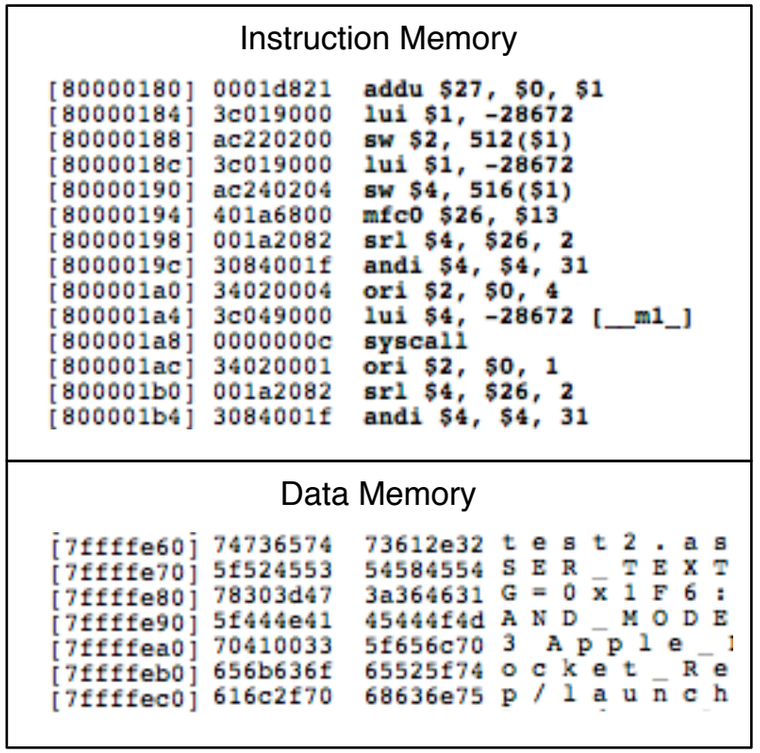


ENIAC

The Stored Program Computer

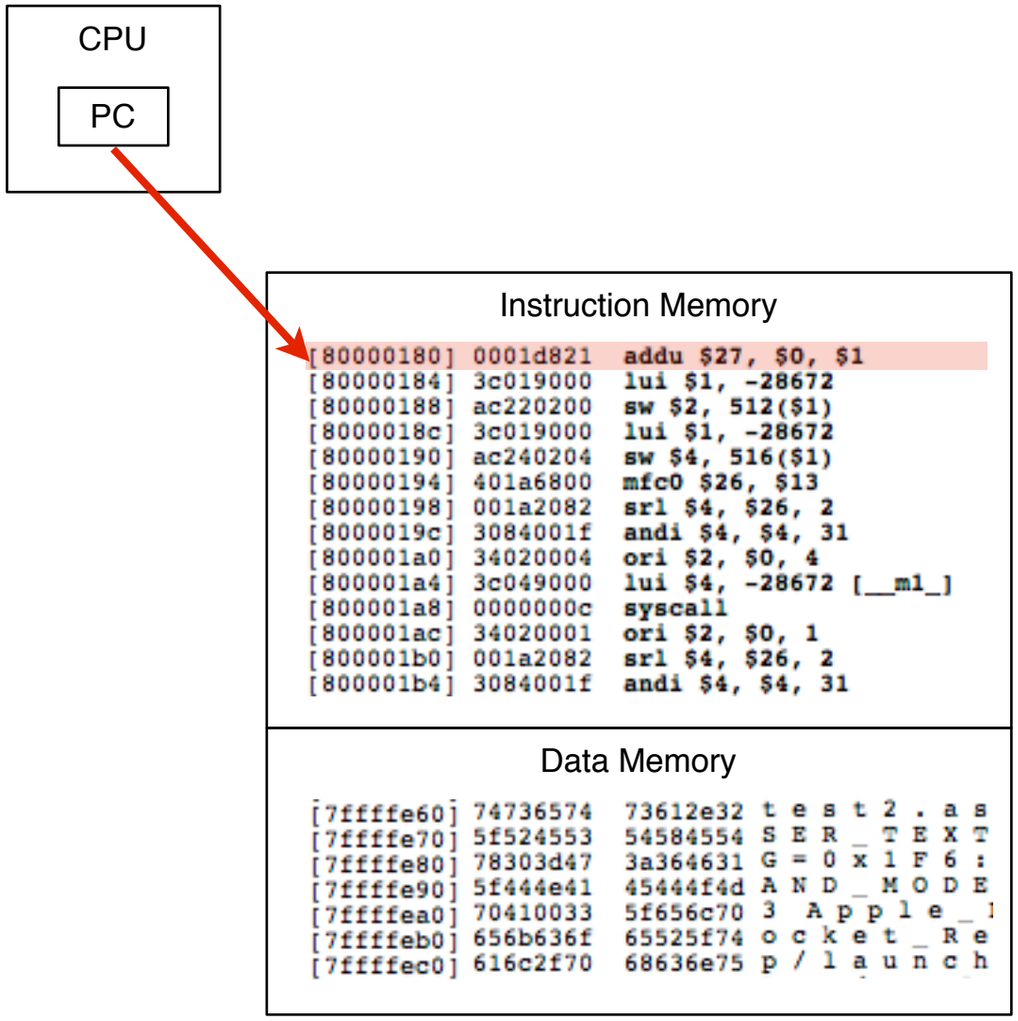
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





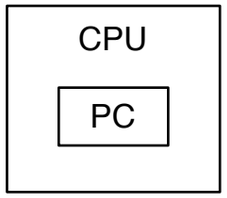
The Stored Program Computer

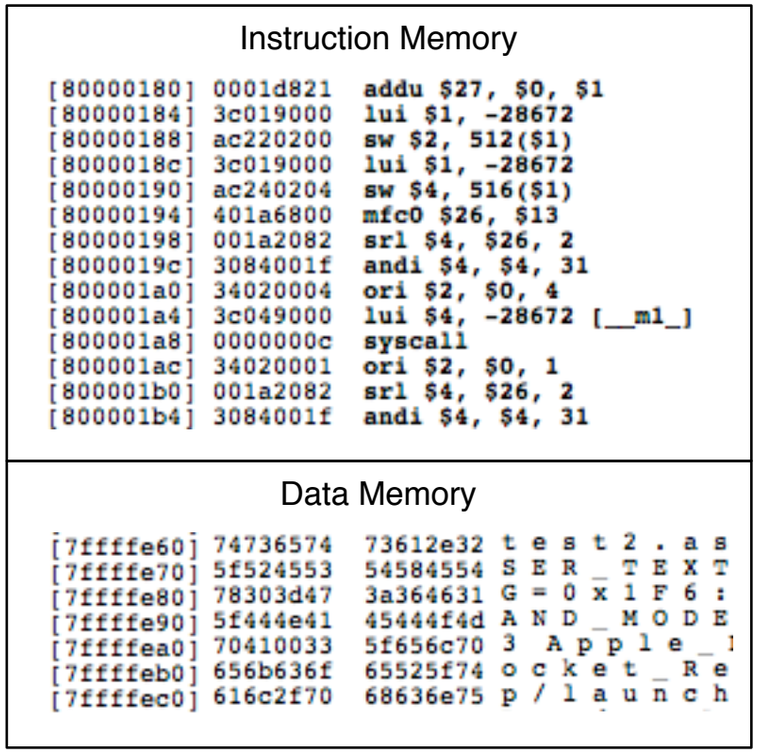
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



The Stored Program Computer

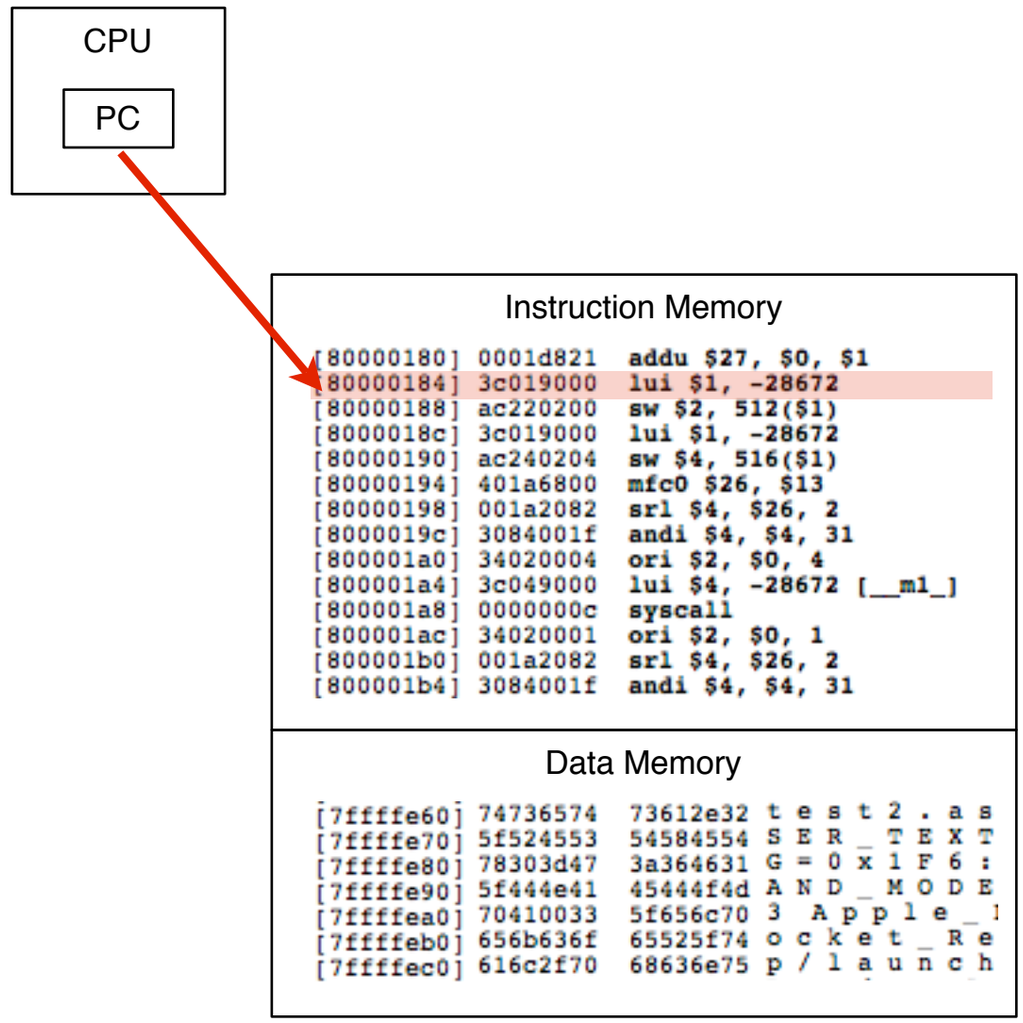
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





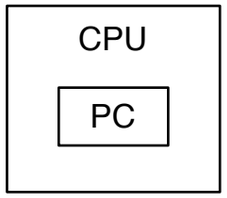
The Stored Program Computer

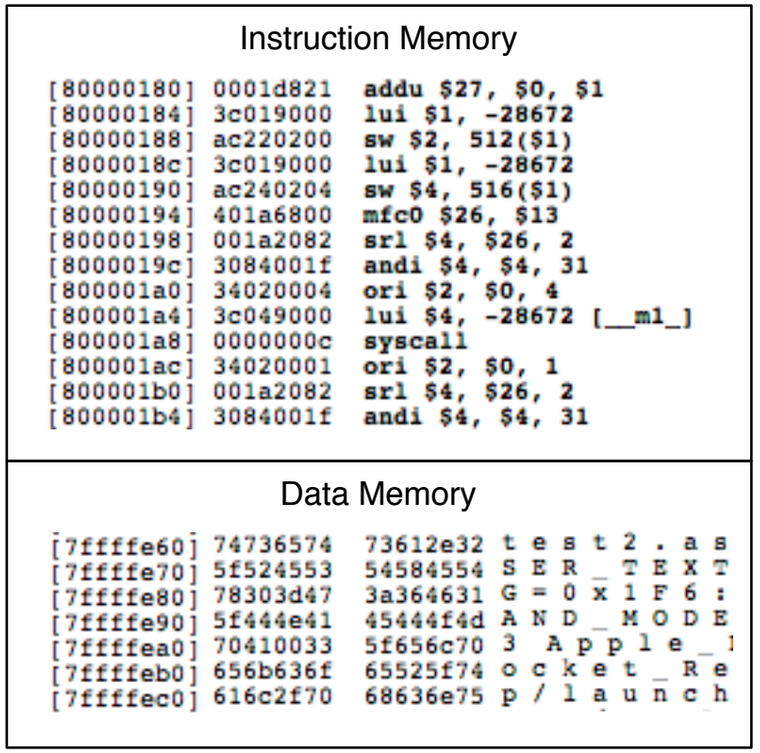
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



The Stored Program Computer

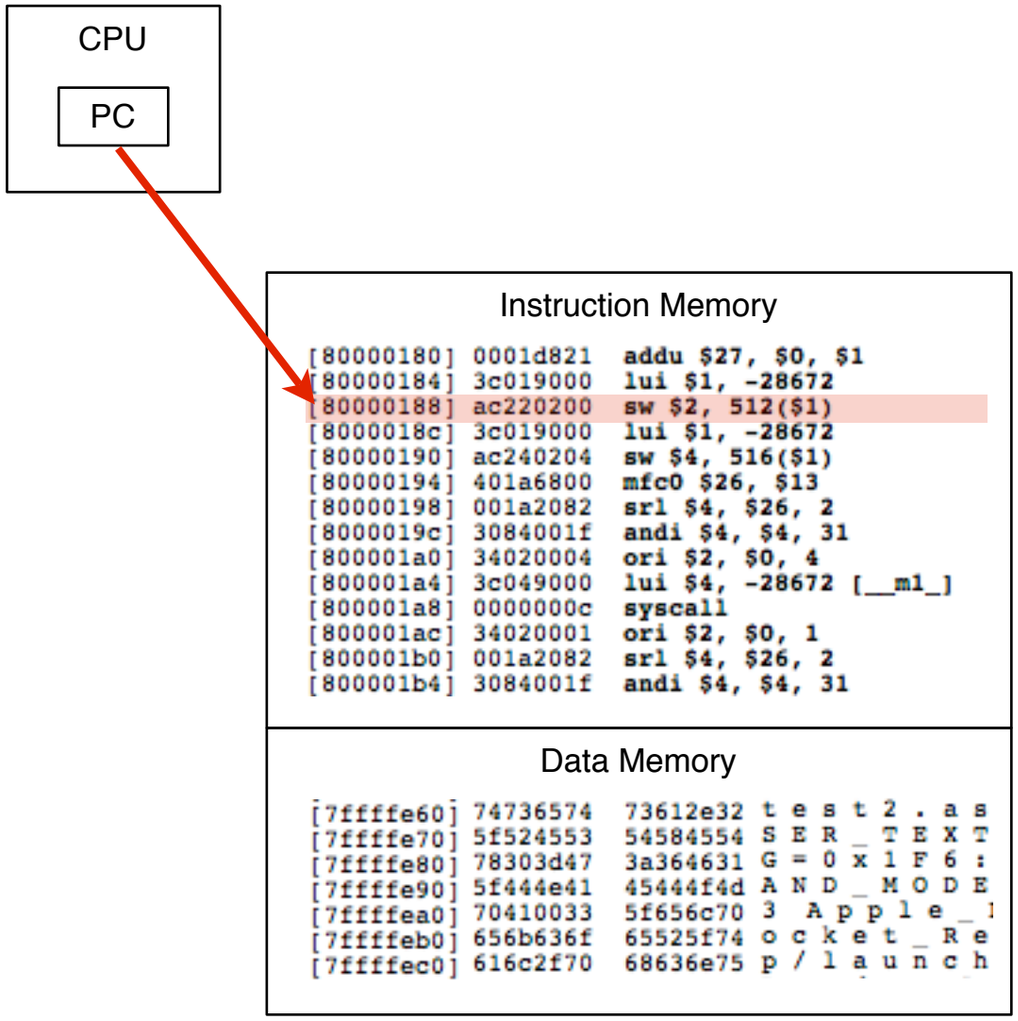
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





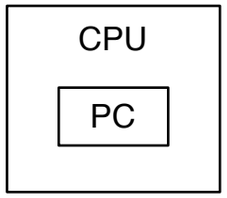
The Stored Program Computer

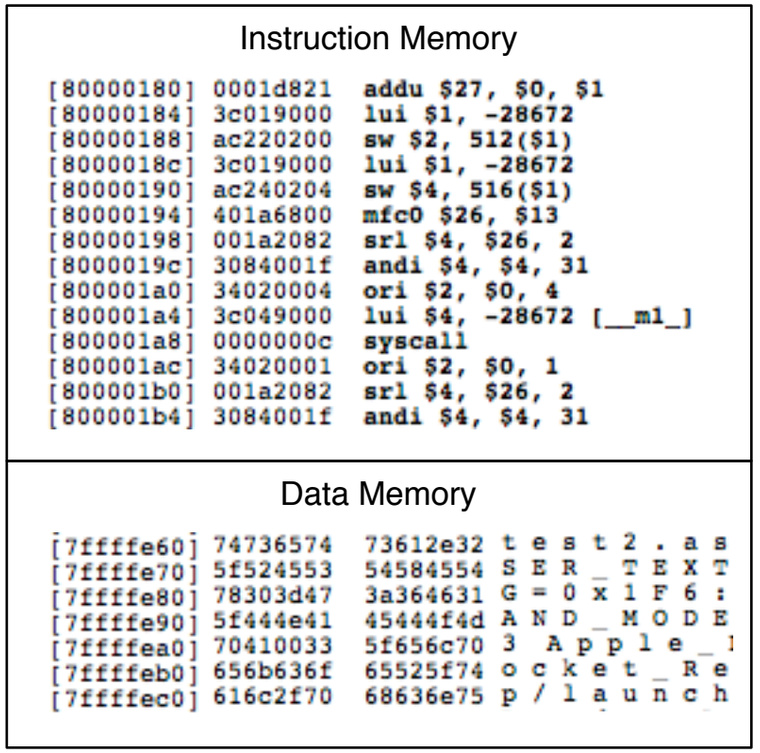
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



The Stored Program Computer

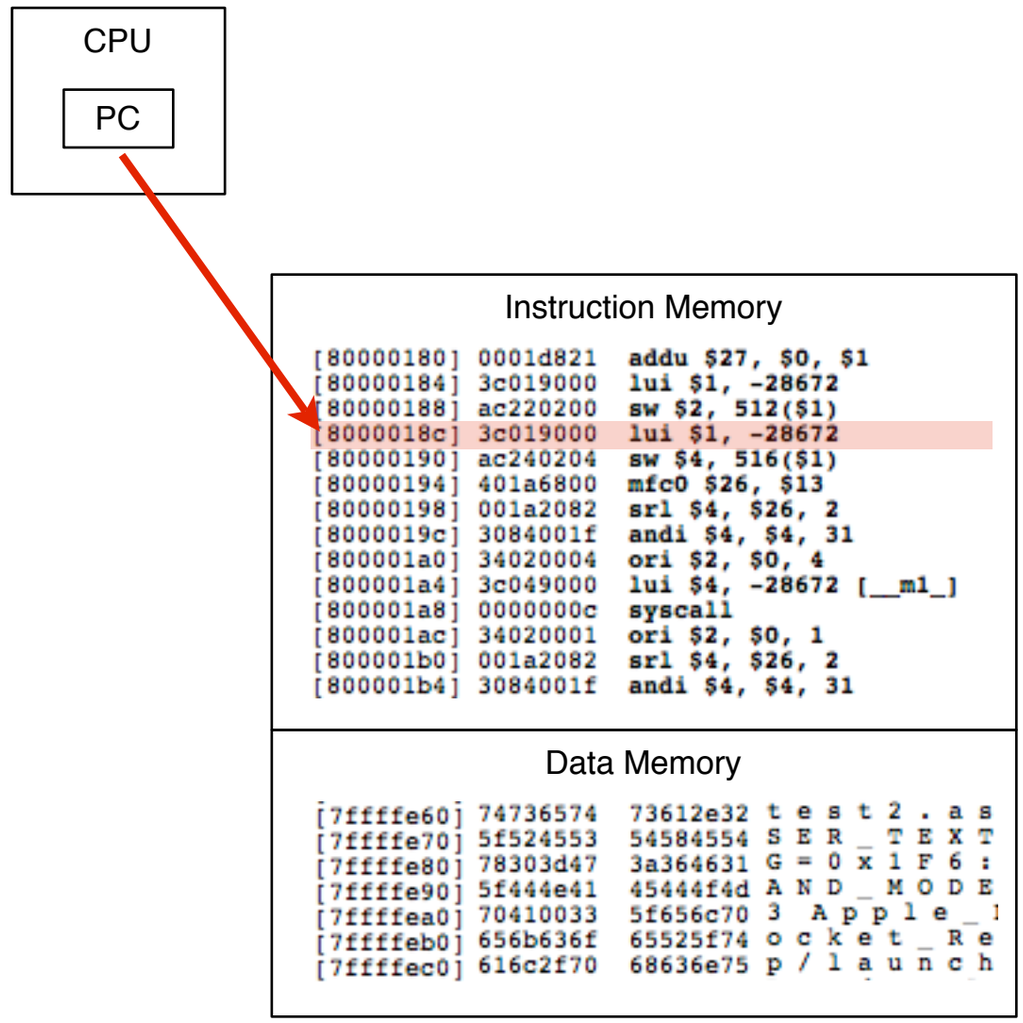
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





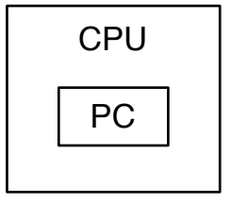
The Stored Program Computer

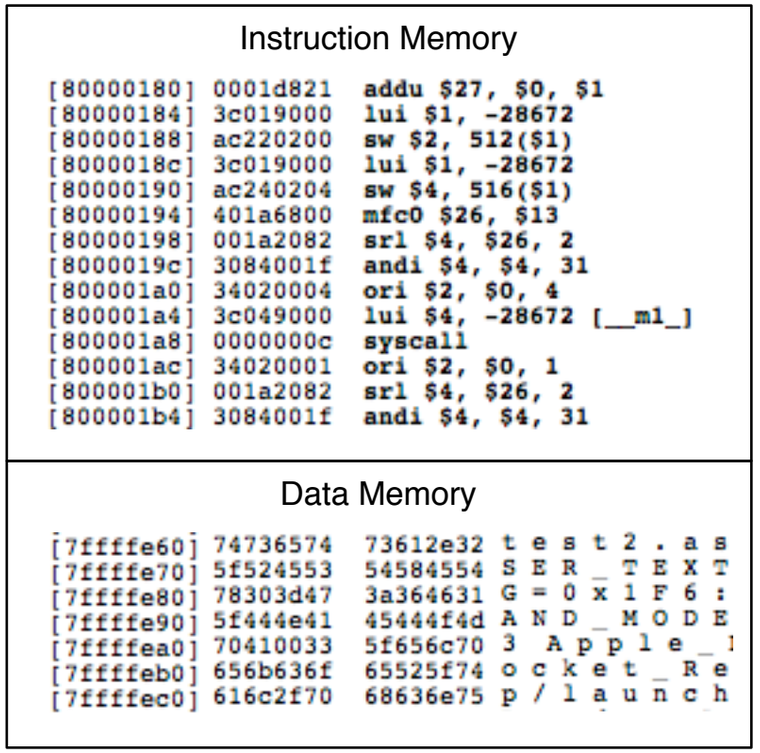
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





The Instruction Set Architecture (ISA)

- The ISA is the set of instructions a computer can execute
- All programs are combinations of these instructions
- It is an abstraction that programmers (and compilers) use to express computations
  - The ISA defines a set of operations, their semantics, and rules for their use.
  - The software agrees to follow these rules.
- The hardware can implement those rules IN ANY WAY IT CHOOSES!
  - Directly in hardware
  - Via a software layer (i.e., a virtual machine)
  - Via a trained monkey with a pen and paper
  - Via a software simulator (like SPIM)
- Also called “the big A architecture”

The MIPS ISA

We Will Study Two ISAs

- MIPS
  - Simple, elegant, easy to implement
  - Designed with the benefit many years ISA design experience
  - Designed for modern programmers, tools, and applications
  - The basis for your implementation project in 141L
  - Not widely used in the real world (but similar ISAs are pretty common, e.g. ARM)
- x86
  - Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  - Designed for 1970s technology
  - Nearly the last in long series of unfortunate ISA designs.
  - The dominant ISA in modern computer systems.

We Will Study Two ISAs

- MIPS
  - Simple, elegant, easy to implement
  - Designed with the benefit many years ISA design experience
  - Designed for modern programmers, tools, and applications
  - The basis for your implementation project in 141L
  - Not widely used in the real world (but similar ISAs are pretty common, e.g. ARM)
- x86
  - Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  - Designed for 1970s technology
  - Nearly the last in long series of unfortunate ISA designs.
  - The dominant ISA in modern computer systems.

You will learn to write MIPS code and implement a MIPS processor

We Will Study Two ISAs

- MIPS
  - Simple, elegant, easy to implement
  - Designed with the benefit many years ISA design experience
  - Designed for modern programmers, tools, and applications
  - The basis for your implementation project in 141L
  - Not widely used in the real world (but similar ISAs are pretty common, e.g. ARM)
- x86
  - Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  - Designed for 1970s technology
  - Nearly the last in long series of unfortunate ISA designs.
  - The dominant ISA in modern computer systems.

You will learn to write MIPS code and implement a MIPS processor

You will learn to read a common subset of x86

MIPS Basics

- Instructions
  - 4 bytes (32 bits)
  - 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  - Instructions operate on memory and registers
- Memory Data types (also aligned)
  - Bytes -- 8 bits
  - Half words -- 16 bits
  - Words -- 32 bits
  - Memory is denote “M” (e.g., M[0x10] is the byte at address 0x10)
- Registers
  - 32 4-byte registers in the “register file”
  - Denoted “R” (e.g., R[2] is register 2)
- There’s a handy reference on the inside cover of your text book and a detailed reference in Appendix B.

Bytes and Words

Byte addresses

Address	Data
0x0000	0xAA
0x0001	0x15
0x0002	0x13
0x0003	0xFF
0x0004	0x76
...	.
0xFFFE	.
0xFFFF	.

Half Word Addrs

Address	Data
0x0000	0xAA15
0x0002	0x13FF
0x0004	.
0x0006	.
...	.
...	.
...	.
0xFFFC	.

Word Addresses

Address	Data
0x0000	0xAA1513FF
0x0004	.
0x0008	.
0x000C	.
...	.
...	.
...	.
0xFFFC	.

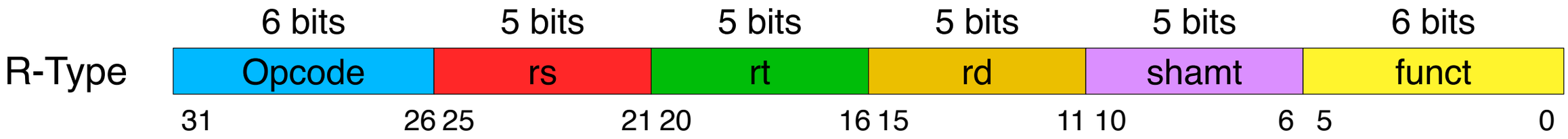
- In modern ISAs (including MIPS) memory is "byte addressable"
- In MIPS, half words and words are aligned.

The MIPS Register File

- All registers are the same
  - Where a register is needed any register will work
- By convention, we use them for particular tasks
  - Argument passing
  - Temporaries, etc.
  - These rules (“the register discipline”) are part of the ISA
- $zero is the “zero register”
  - It is always zero.
  - Writes to it have no effect.

Name	number	use	Callee saved
$zero	0	zero	n/a
$at	1	Assemble Temp	no
$v0 - $v1	2 - 3	return value	no
$a0 - $a3	4 - 7	arguments	no
$t0 - $t7	8 - 15	temporaries	no
$s0 - $s7	16 - 23	saved temporaries	yes
$t8 - $t9	24 - 25	temporaries	no
$k0 - $k1	26 - 27	Res. for OS	yes
$gp	28	global ptr	yes
$sp	29	stack ptr	yes
$fp	30	frame ptr	yes
$ra	31	return address	yes

MIPS R-Type Arithmetic Instructions

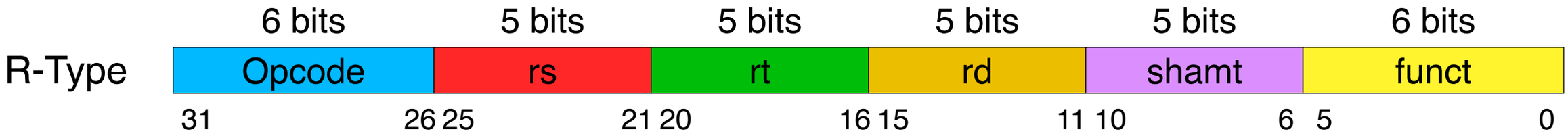


- R-Type instructions encode operations of the form "a = b OP c" where 'OP' is +, -, <<, &, etc.
  - More formally, R[rd] = R[rs] OP R[rt]
- Bit fields
  - "opcode" encodes the operation type.
  - "funct" specifies the particular operation.
  - "rs" are "rt" source registers; "rd" is the destination register
    - 5 bits can specify one of 32 registers.
- "shamt" is the "shift amount" for shift operations
  - Since registers are 32 bits, 5 bits are sufficient

Examples

- `add $t0, $t1, $t2`
  - `R[8] = R[9] + R[10]`
  - `opcode = 0, funct = 0x20`
- `nor $a0, $s0, $t4`
  - `R[4] = ~(R[16] | R[12])`
  - `opcode = 0, funct = 0x27`
- `sll $t0, $t1, 4`
  - `R[4] = R[16] << 4`
  - `opcode = 0, funct = 0x0, shamt = 4`

MIPS R-Type Control Instructions

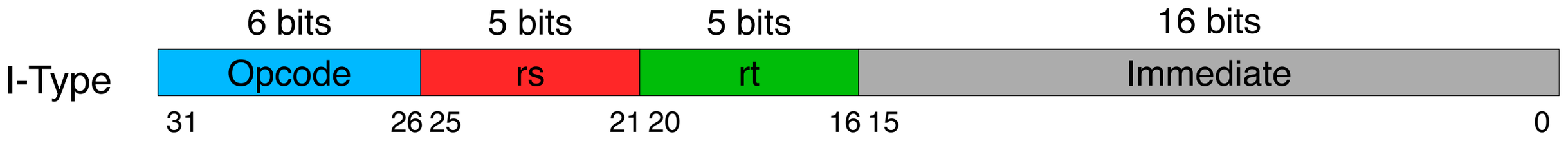


- R-Type encodes “register-indirect” jumps
- Jump register
  - jr rs: PC = R[rs]
- Jump and link register
  - jalr rs, rd: R[rd] = PC + 8; PC = R[rs]
  - rd default to $ra (i.e., the assembler will fill it in if you leave it out)

Examples

- `jr $t2`
  - `PC = r[10]`
  - `opcode = 0, funct = 0x8`
- `jalr $t0`
  - `PC = R[8]`
  - `R[31] = PC + 8`
  - `opcode = 0, funct = 0x9`
- `jalr $t0, $t1`
  - `PC = R[8]`
  - `R[9] = PC + 8`
  - `opcode = 0, funct = 0x9`

MIPS I-Type Arithmetic Instructions

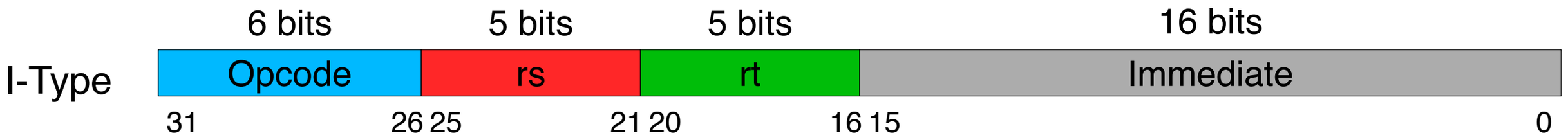


- I-Type arithmetic instructions encode operations of the form “a = b OP #”
- ‘OP’ is +, -, <<, &, etc and # is an integer constant
  - More formally, e.g.: R[rd] = R[rs] + 42
- Components
  - “opcode” encodes the operation type.
  - “rs” is the source register
  - “rd” is the destination register
- “immediate” is a 16 bit constant used as an argument for the operation

Examples

- `addi $t0, $t1, -42`
  - `R[8] = R[9] + -42`
  - `opcode = 0x8`
- `ori $t0, $zero, 42`
  - `R[4] = R[0] | 42`
  - `opcode = 0xd`
  - `Loads a constant into $t0`

MIPS I-Type Branch Instructions

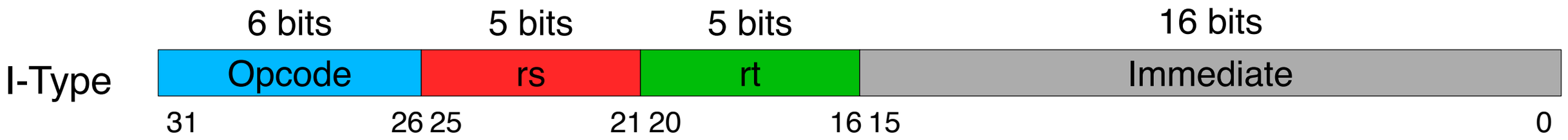


- I-Type also encode branches
  - if (R[rd] OP R[rs])
    PC = PC + 4 + 4 * Immediate
    else
    PC = PC + 4
- Components
  - "rs" and "rt" are the two registers to be compared
  - "rt" is sometimes used to specify branch type.
- "immediate" is a 16 bit branch offset
  - It is the signed offset to the target of the branch
  - Limits branch distance to 32K instructions
  - Usually specified as a label, and the assembler fills it in for you.

Examples

- `beq $t0, $t1, -42`
  - `if R[8] == R[9]`
    `PC = PC + 4 + 4*-42`
  - `opcode = 0x4`
- `bgez $t0, -42`
  - `if R[8] >= 0`
    `PC = PC + 4 + 4*-42`
  - `opcode = 0x1`
  - `rt = 1`

MIPS I-Type Memory Instructions

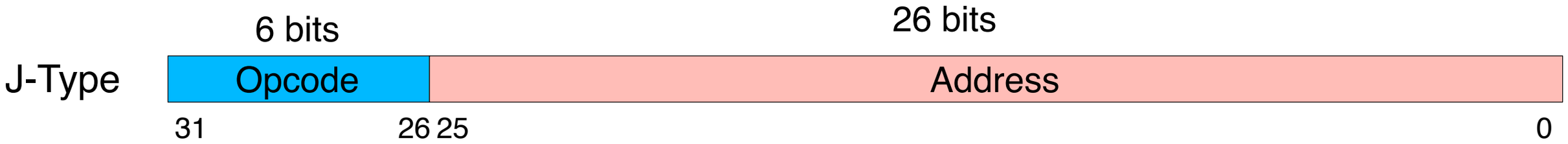


- I-Type also encode memory access
  - Store: M[R[rs] + Immediate] = R[rt]
  - Load: R[rt] = M[R[rs] + Immediate]
- MIPS has load/stores for byte, half word, and word
- Sub-word loads can also be signed or unsigned
  - Signed loads sign-extend the value to fill a 32 bit register.
  - Unsigned zero-extend the value.
- "immediate" is a 16 bit offset
  - Useful for accessing structure components
  - It is signed.

Examples

- `lw $t0, 4($t1)`
  - `R[8] = M[R[9] + 4]`
  - `opcode = 0x23`
- `sb $t0, -17($t1)`
  - `M[R[12] + -17] = R[4]`
  - `opcode = 0x28`

MIPS J-Type Instructions

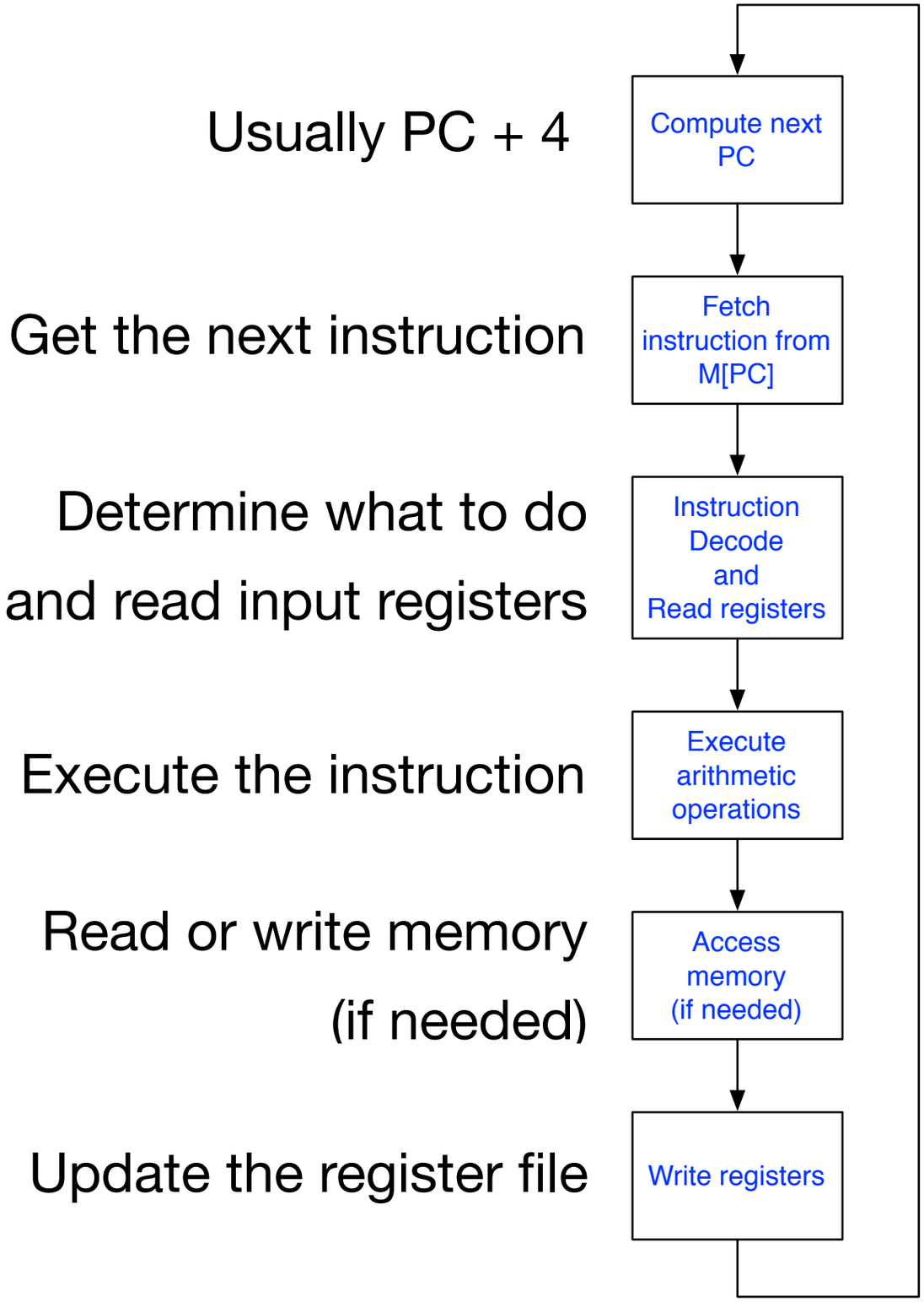


- J-Type encodes the jump instructions
- Plain Jump
  - JumpAddress = {PC+4[31:28],Address,2'b0}
  - Address replaces *most* of the PC
  - PC = JumpAddress
- Jump and Link
  - R[$ra] = PC + 8; PC = JumpAddress;
- J-Type also encodes misc instructions
  - syscall, interrupt return, and break (more later)

Examples

- `j $t0`
  - `PC = R[8]`
  - `opcode = 0x2`
- `jal $t0`
  - `R[31] = PC + 8`
  - `PC = R[8]`

Executing a MIPS program



- All instructions have
  - <= 1 arithmetic op
  - <= 1 memory access
  - <= 2 register reads
  - <= 1 register write
  - <= 1 branch
- All instructions go through all the steps
- As a result
  - Implementing MIPS is (sort of) easy!
  - The resulting HW is (relatively) simple!

MIPS Mystery 1: Delayed Loads

- The value retrieved by a load is not available to the next instruction.

Example

```
ori $t0, $zero, 4
sw  $t0, 0($sp)
lw  $t1, 0($sp)
or  $t2, $t1, $zero
or  $t3, $t1, $zero
```

```
$t2 == 0
$t3 == 4
```

file: delayed_load.s

MIPS Mystery 1: Delayed Loads

- The value retrieved by a load is not available to the next instruction.

Example

```
ori $t0, $zero, 4
sw  $t0, 0($sp)
lw  $t1, 0($sp)
or  $t2, $t1, $zero
or  $t3, $t1, $zero
```

```
$t2 == 0
$t3 == 4
```

file: delayed_load.s

Why? We'll talk about it in a few weeks.

MIPS Mystery 2: Delayed Branches

- The instruction after the branch executes even if the branch is taken.
- All jumps and branches are delayed -- the next instruction *always* executes

Example

```
ori $t0, $zero, 4
beq $t0, $t0, foo
ori $t0, $zero, 5
foo:

        $t0 == 5
```

file: delayed_branch.s

MIPS Mystery 2: Delayed Branches

- The instruction after the branch executes even if the branch is taken.
- All jumps and branches are delayed -- the next instruction *always* executes

Why? We'll talk about it in a few weeks.

Example

```
ori $t0, $zero, 4
beq $t0, $t0, foo
ori $t0, $zero, 5
foo:

        $t0 == 5
```

file: delayed_branch.s

Quiz 1

- Why are you here? What's your major?
  - I've wanted to write an operating system since I was a little kid and designing a processor to go with it sounds cool too.
  - [I'm majoring in] Computer Science. I enjoy programming and making tools for humintarian aid. I also find this field to be very fascinating and beautiful. I could go on but I'm running out of time...
  - …I was always very interested in computers ever since I was young. Plus, the average salary for us is pretty decent!
  - I am a double major in Physics and Computer Science. I'll be attending graduate school in Physics, and my research interests are in Computational Astrophysics.
  - To gain an adequate understanding of processor & ISA design and implementation, but admittedly primarily for the purpose of fulfilling academic course requirements.
  - Computer Science, because I thought programming was cool. Then I found out the truth and now I'm too committed to change.
  - Computer Science BS, Psychology BA. I transferred to do psychology, realized the psych program here was …lame …, got bored quickly … got completely hooked on CSE.

Question 3: Multiple Answer

Average Score 0.49206 points

Which of the following statements are true about instruction set architectures?

Correct	Answers	Percent Correct	Percent Incorrect
☑	The ISA defines a contract between the program and the hardware.	67.692%	32.308%
	The ISA determines how the processor will implement the operations the ISA defines.	72.308%	27.692%
☑	The ISA defines a set of basic operations that the processor can perform.	98.462%	1.538%
	The ISA that a computer uses determines which programming languages can be used to program the computer.	93.846%	6.154%

Question 3: Multiple Answer

Average Score 0.49206 points

Which of the following statements are true about instruction set architectures?

Correct	Answers	Percent Correct	Percent Incorrect
☑	The ISA defines a contract between the program and the hardware.	67.692%	32.308%
	The ISA determines how the processor will implement the operations the ISA defines.	72.308%	27.692%
☑	The ISA defines a set of basic operations that the processor can perform.	98.462%	1.538%
	The ISA that a computer uses determines which programming languages can be used to program the computer.	93.846%	6.154%

Question 4: Multiple Answer

Average Score 0.8125 points

Which of the following statement are true about the MIPS ISA?

Correct	Answers	Percent Correct	Percent Incorrect
	MIPS is the dominant ISA in use in computers today.	98.462%	1.538%
	MIPS instructions are 8 bytes long.	98.462%	1.538%
☑	MIPS instructions can read up to 2 register values from the register file and write at most 1 register.	89.231%	10.769%
☑	Regardless of instruction type, the opcode is always in the same location in MIPS instructions.	92.308%	7.692%
	MIPS is more complex than the x86 ISA.	100%	0%

Question 5: True/False

Average Score 0.52381 points

Moore's Law specifies that processor performance double every 18-24 months.

Correct	Answers	Percent Answered
	True	49.231%
☑	False	50.769%
	Unanswered	0%

Question 6: Multiple Choice

Average Score 0.93651 points

Performance scaling of microprocessors slowed around 2004 for which reason?

Correct		Percent Answered
	Moore's Law ended.	3.077%
☑	Power constraints caused the end of clock speed scaling.	92.308%
	Users stopped being interested in processor performance.	0%
	Chip manufacturers started building chip-multiprocessor because they were easier to program.	1.538%
	Unanswered	3.077%

Question 7: Multiple Answer

Average Score 0.7619 points

Which of the following are key components of a stored program computer?

Correct	Answers	Percent Correct	Percent Incorrect
☑	Memory.	87.692%	12.308%
	IO devices.	81.538%	18.462%
☑	The program counter.	86.154%	13.846%
	A register file.	64.615%	35.385%
	Reconfigurable hardware.	90.769%	9.231%
	The ability to execute java programs.	96.923%	3.077%
☑	A processor that executes a series of discrete instructions.	83.077%	16.923%

Question 8: Multiple Choice

Average Score 1.11111 points

What will the values of $t2 and $t3 be after executing this MIPS code?

```
 ori $t0, $zero, 4
 sw  $t0, 0($sp)
 beq $t0, $t0, foo
 lw  $t1, 0($sp)
 ori $t2, $zero, 0
foo:
 or  $t2, $t1, $zero
 or  $t3, $t1, $zero
```

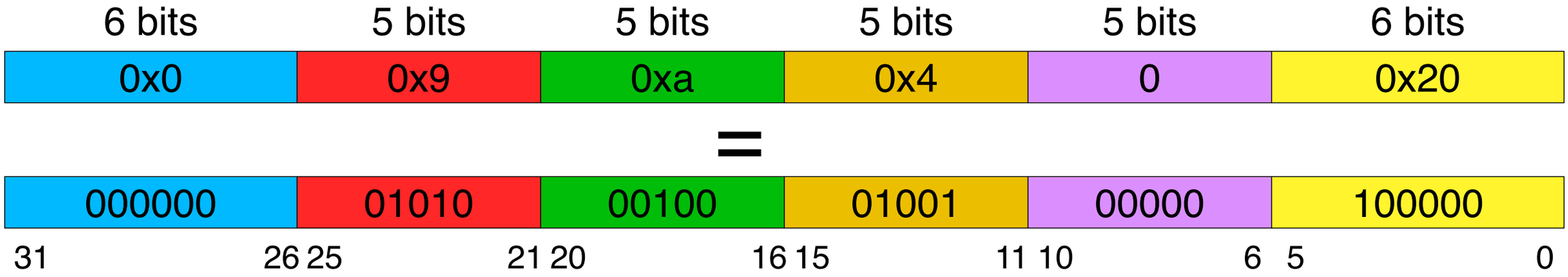
Correct		Percent Answered
	$t2 = 0 $t3 = 0	13.846%
☑	$t2 = 0 $t3 = 4	53.846%
	$t2 = 4 $t3 = 0	1.538%
	$t2 = 4 $t3 = 4	27.692%
	None of the above.	1.538%
	Unanswered	1.538%

Live Demo!

Source code available on the course web site

Example 1: add.s

addr	inst bits	inst	source code
[00400000]	01444820	**add $9, $10, $4**	*; 2: add $t1, $t2, $a0*



Example: Warts

- Files
  - delayed_branch.s
  - delayed_load.s
- Make sure to set SPIM settings to “bare machine”
  - See the SPIM tutorial
  - Always check that you’ve got this set. We will not be using “simple machine” in this class.

Example: conditional.s

```
i = 42
if (i & 7)
  i += 8
else
  i += 4
```

```
    ori  $t0, $zero, 42
    andi $t1, $t0, 7
    beq  $t1, $zero, ifcode
    add $zero, $zero, $zero
elsecode:
    addi $t0, $t0, 4
    beq $zero, $zero, followon
    add $zero, $zero, $zero
ifcode:
    addi $t0, $t0, 8
followon:
```

Branch Delay Slots

$t0 is i

Example: loop.s

```
i = 5
do
  j += i
  i--;
while i != 0
```

$t0 is i
$t1 is j

```
[00400000] 34080005  ori $8, $0, 5              ; 1: ori $t0, $zero, 5
[00400004] 01284820  add $9, $9, $8             ; 3: add $t1, $t1, $t0
[00400008] 2108ffff  addi $8, $8, -1            ; 4: addi $t0, $t0, -1
[0040000c] 1500fffe  bne $8, $0, -8 [top-0x0040000c]; 5: bne $t0, $zero, top
[00400010] 00000020  add $0, $0, $0             ; 6: add $zero, $zero, $zero #noop in
the branch delay slot.
```

Function Calls

- Challenges
  - Passing in i and calling lg
  - Returning the sum
  - Continuing execution after the call
  - Allocating temporaries
  - Releasing temporaries

Example

```
int lg(int i) {
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
}
```

Calling and Returning

- Passing arguments
  - The first 4 in $a0...$a3
  - Any more go on the stack
- Invoking the function
  - jal <label>
  - Stores PC + 8 in $ra
- Return value in $v0
- Return to caller
  - jr $ra

Example

```
ori $a0, $zero, 4
jal log2
addi $zero, $zero, 0
... access $v0 ...
log2:
...
ori $v0, $zero, 0
jr $ra
```

Managing Registers

- Sharing registers
  - A called function will modify registers
  - The caller needs to keep some values around.
- The ISA specifies which registers a function can modify
- A function can use “callee-saved” registers, but must restore their value.

Name	number	use	Callee saved
$zero	0	zero	n/a
$at	1	Assemble Temp	no
$v0 - $v1	2 - 3	return value	no
$a0 - $a3	4 - 7	arguments	no
$t0 - $t7	8 - 15	temporaries	no
$s0 - $s7	16 - 23	saved temporaries	yes
$t8 - $t9	24 - 25	temporaries	no
$k0 - $k1	26 - 27	Res. for OS	yes
$gp	28	global ptr	yes
$sp	29	stack ptr	yes
$fp	30	frame ptr	yes
$ra	31	return address	yes

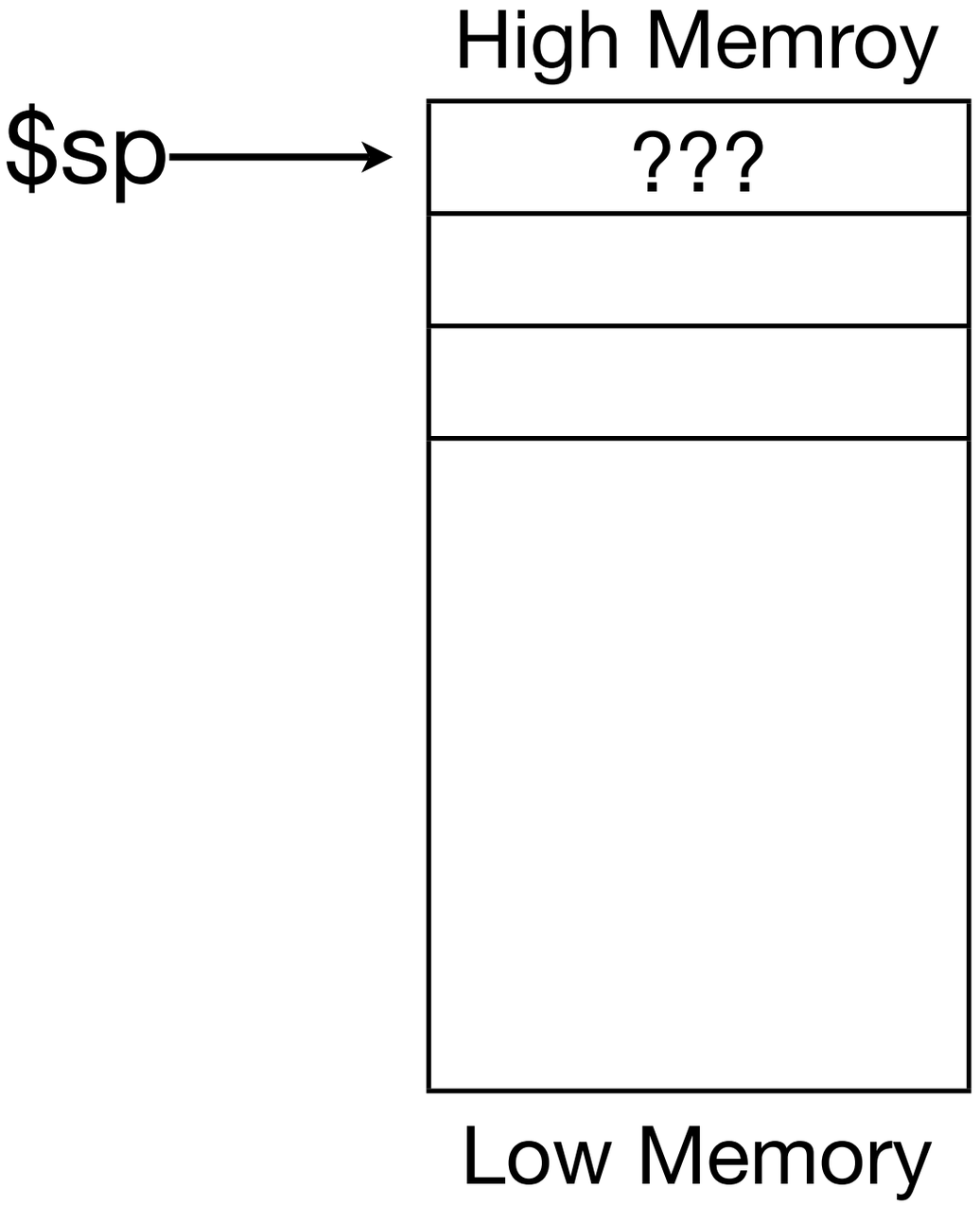
The Stack

- The stack provides local storage for function calls (e.g., for preserving registers)
  - Local variables
  - Register overflow
  - Preserved register contents
- It is as first-in-last-out (FILO) queue
- For historical the stack grows down from high memory addresses to low.
- The stack pointer ($sp) points to the “top” of the stack.

Preserving Registers

Assume $ra = 0xBEEF

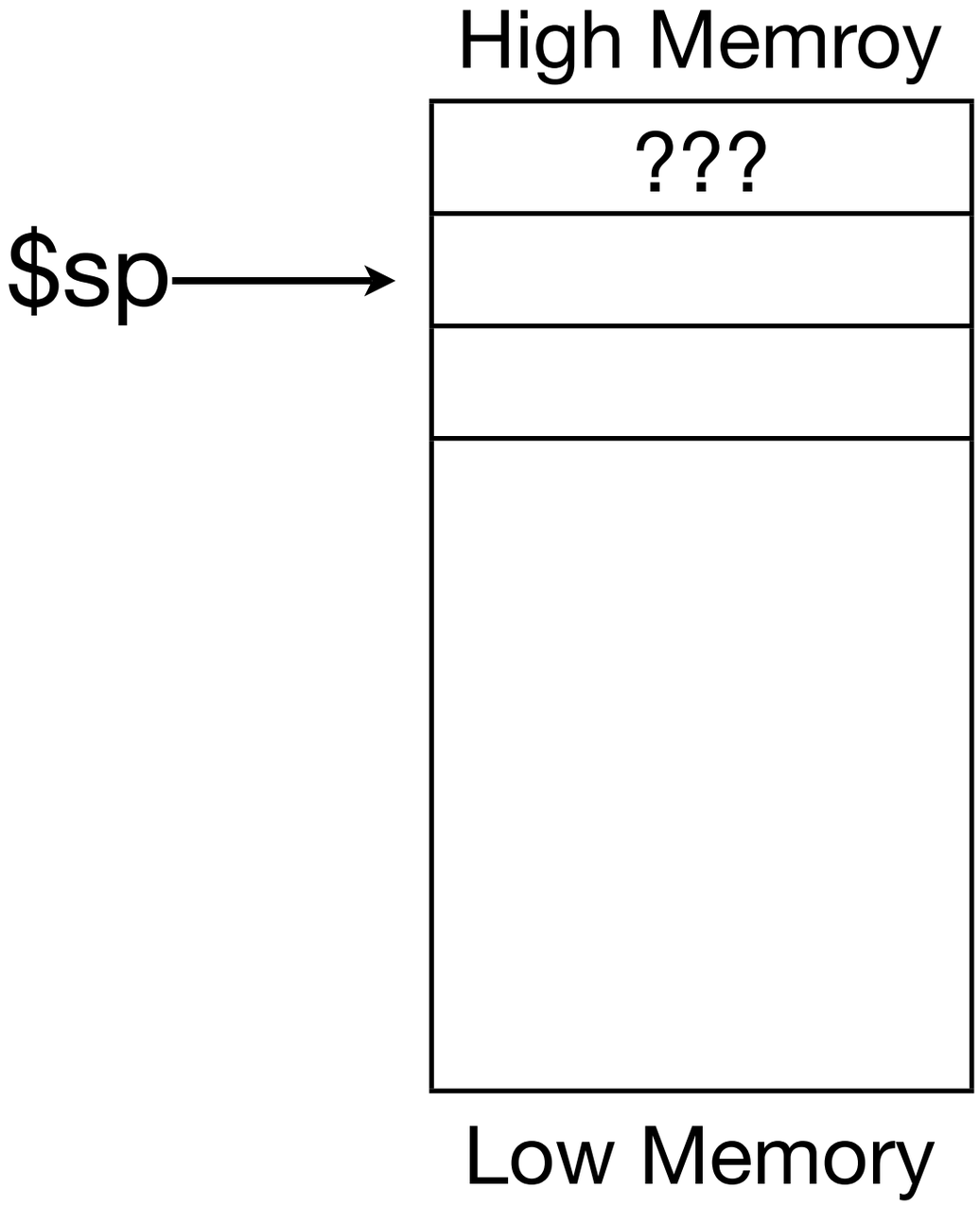
```
To save $ra:
  addi $sp, $sp, -4
  sw $ra, 0($sp)
... function calls ...
To restore $ra:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
```



Preserving Registers

Assume $ra = 0xBEEF

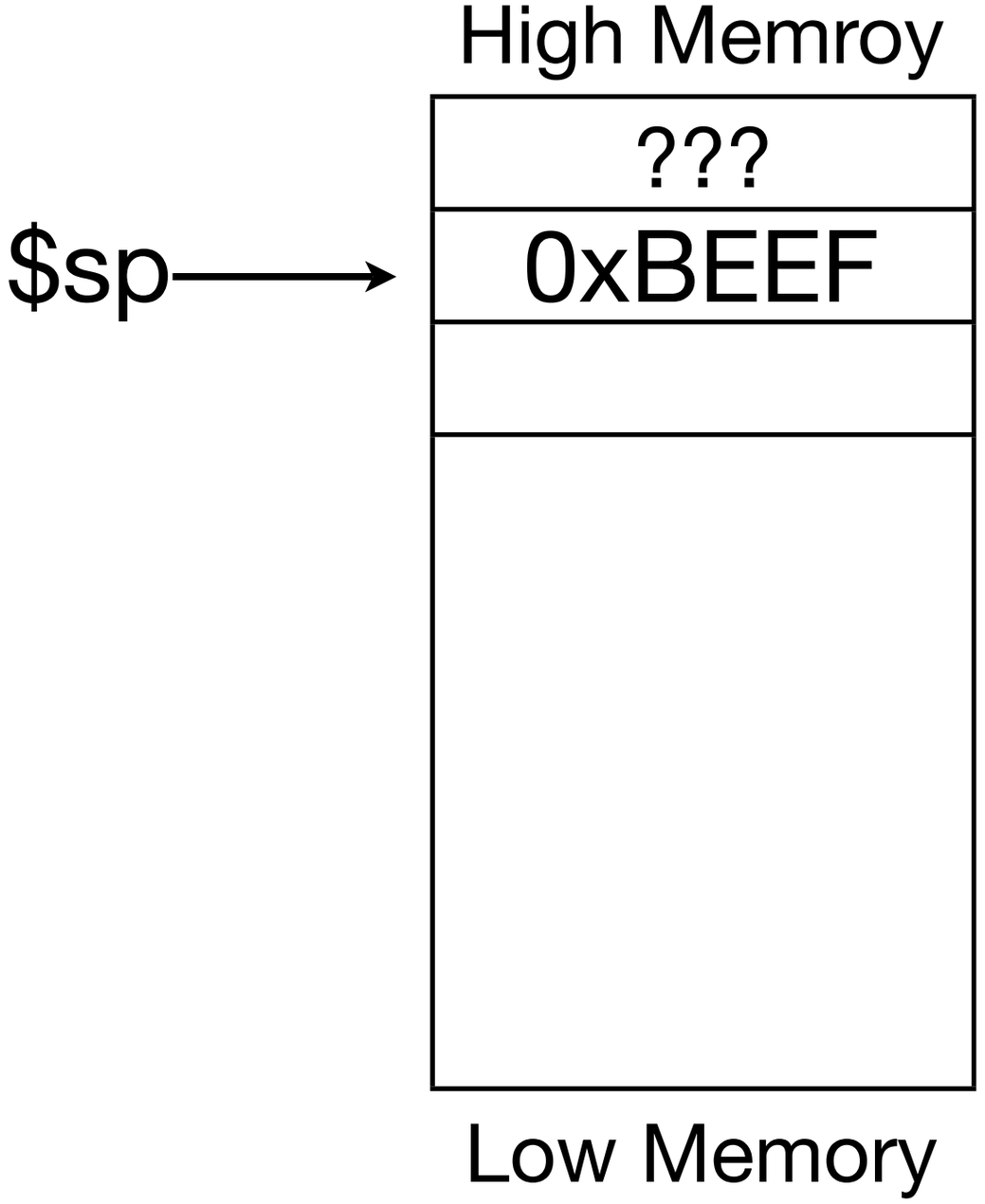
```
To save $ra:
  addi $sp, $sp, -4
  sw $ra, 0($sp)
    ... function calls ...
To restore $ra:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
```



Preserving Registers

Assume $ra = 0xBEEF

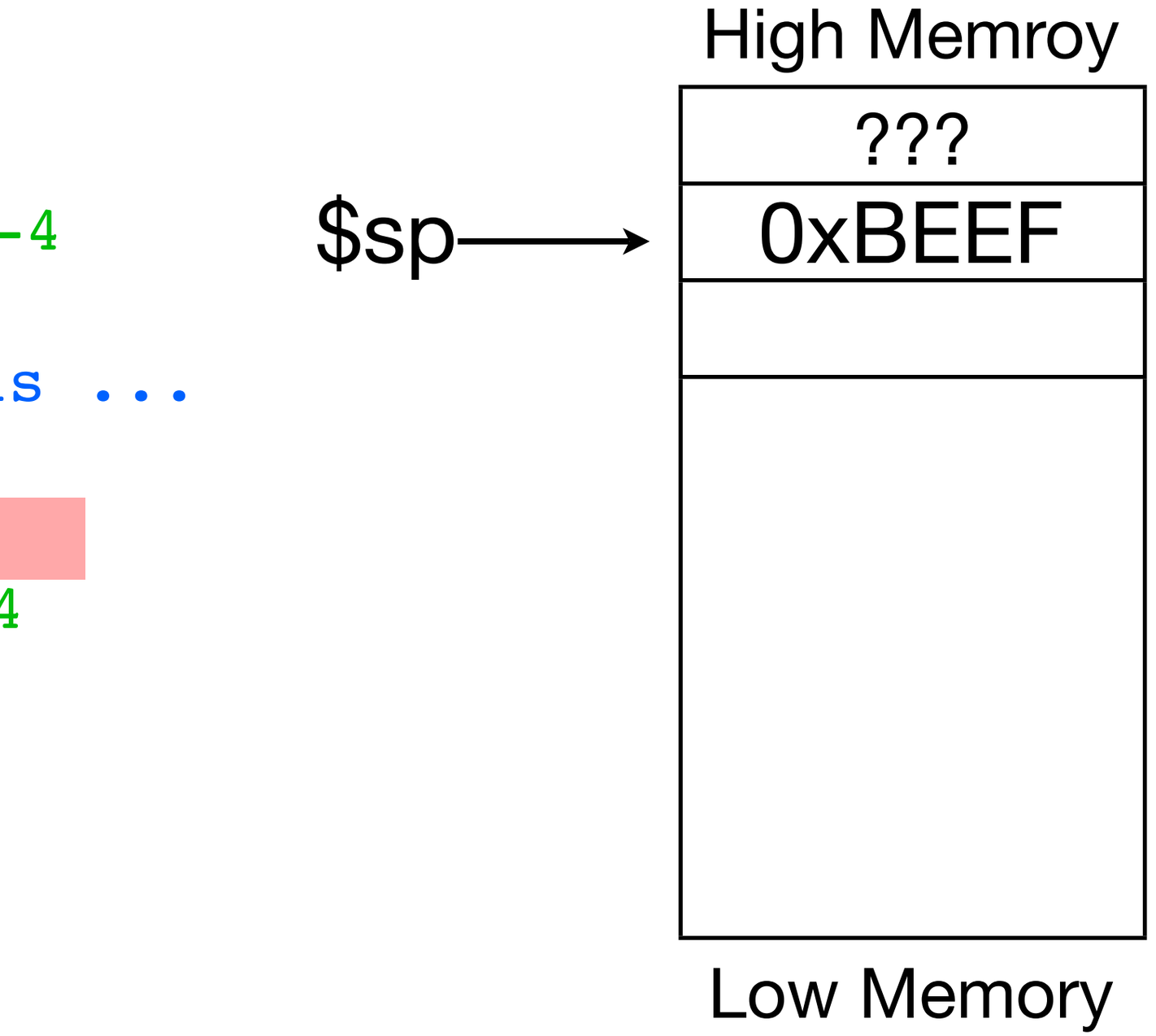
```
To save $ra:
  addi $sp, $sp, -4
  sw $ra, 0($sp)
    function calls ...
... function calls ...
To restore $ra:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
```



Preserving Registers

Assume $ra = 0xBEEF

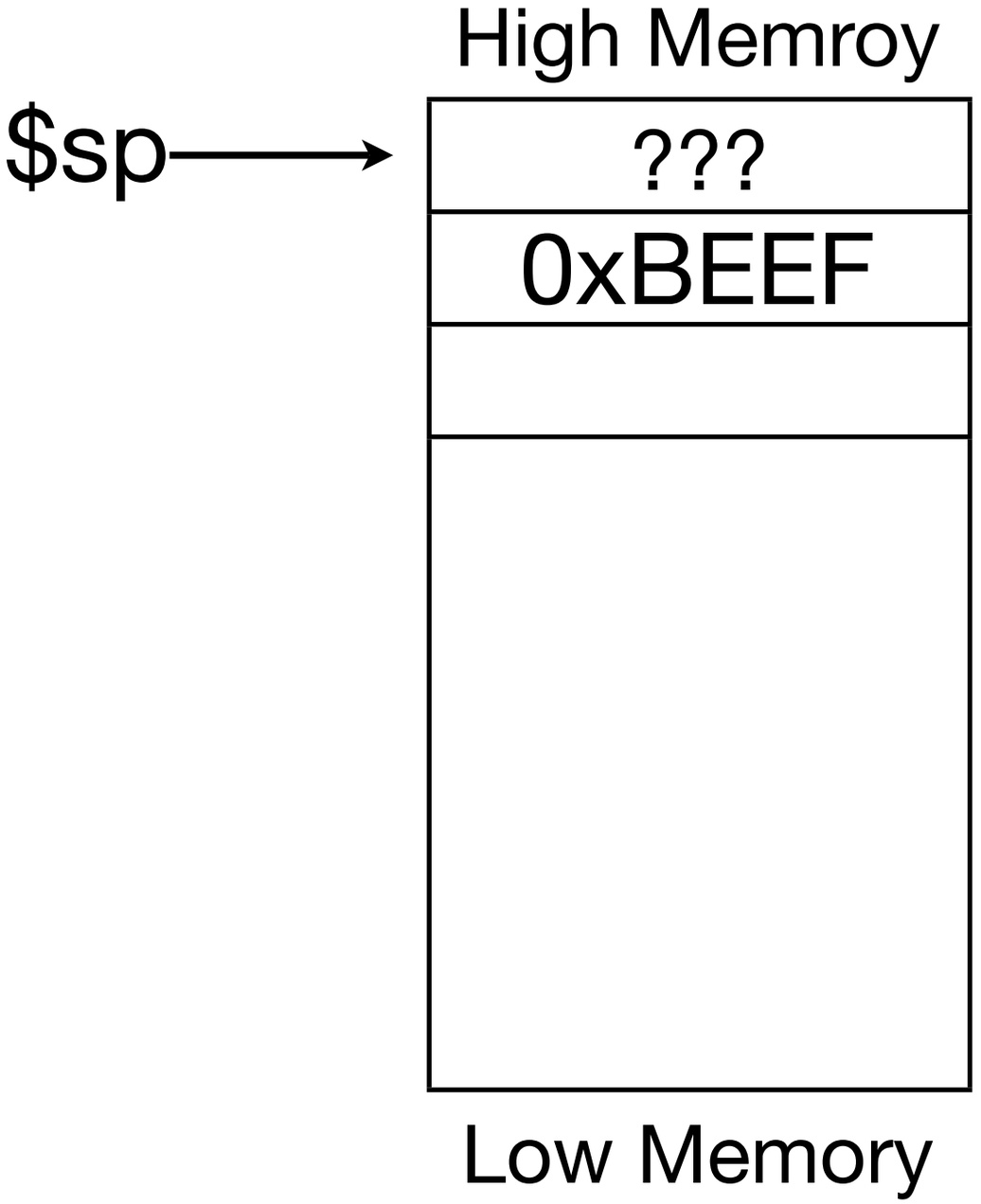
```
To save $ra:
  addi $sp, $sp, -4
  sw $ra, 0($sp)
    ... function calls ...
To restore $ra:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
```



Preserving Registers

Assume $ra = 0xBEEF

```
To save $ra:
  addi $sp, $sp, -4
  sw $ra, 0($sp)
    ... function calls ...
To restore $ra:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
```

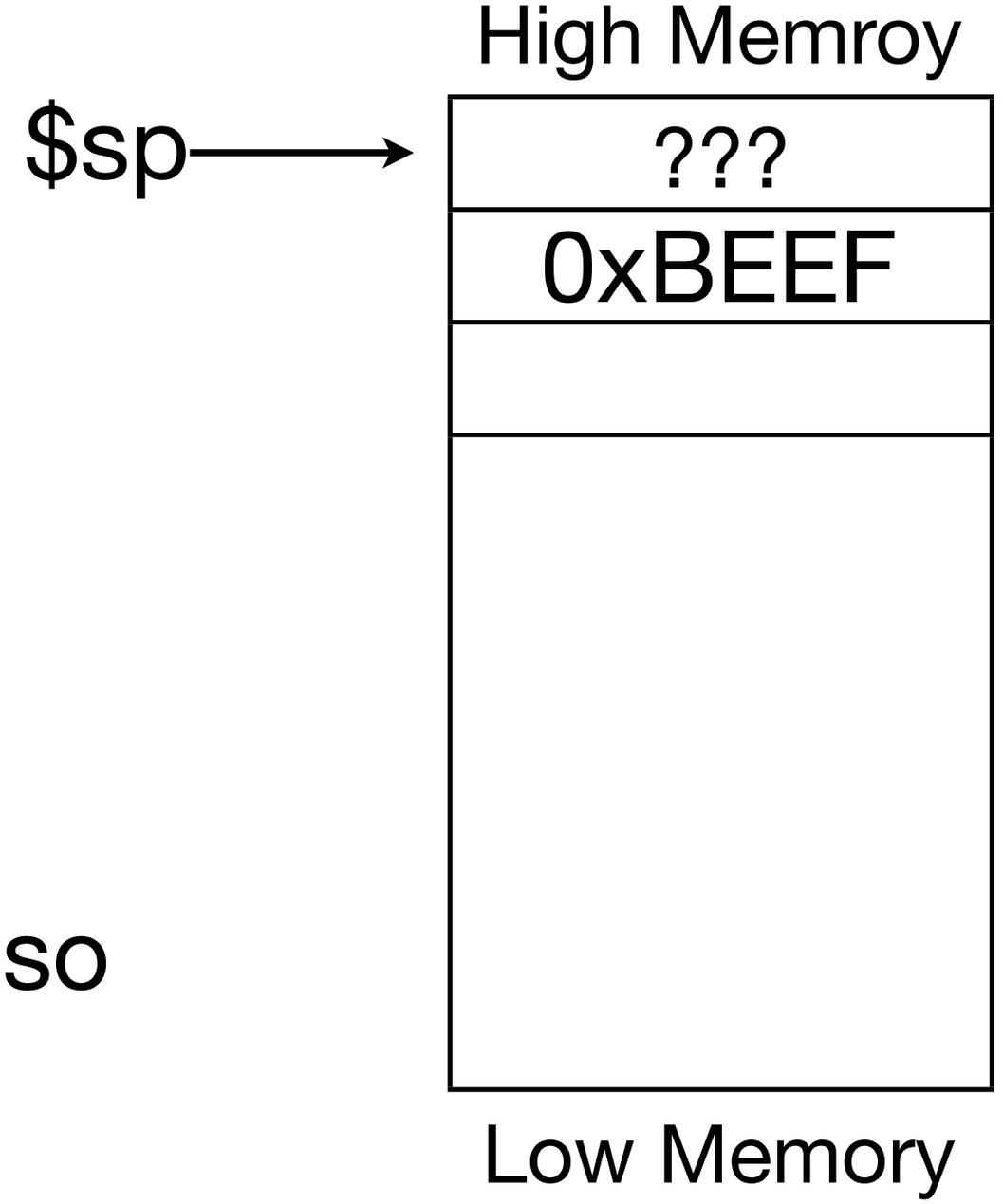


Preserving Registers

Assume $ra = 0xBEEF

```
To save $ra:
  addi $sp, $sp, -4
  sw $ra, 0($sp)
... function calls ...
To restore $ra:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
```

Note that $sp is also restored



```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

Delay slots

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
      lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

```
lg:
    addi $sp, $sp, -4
    sw  $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl  $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```

```
int lg(int i) {
  // Save registers
  if (i)
    return
       lg(i >> 1) + 1;
  else
    return 0;
  // Restore registers
}
```

Delay slots

Live Demo!

Source code available on the class web site

Slides/01 ISA Part-I examples/release/lg.s
Slides/01 ISA Part-I examples/release/lg.c
Slides/01 ISA Part-I examples/release/lg-opt.s

Filling Delay Slots

- Compilers put useful instructions in delay slots.
- Branch delay
  - Use instructions from before the branch.
- Load delay
  - Use an instruction that doesn't need the loaded value
  - Or that needs the old value of the register

```
lg:
    addi $sp, $sp, -4
    bne $a0, $zero, big
    sw  $ra, 0($sp)
    j end
    ori $v0, $zero, 0
big:
    jal lg
    srl  $a0, $a0, 1
    addi $v0, $v0, 1
end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
    add $zero, $zero, $zero
```

Filling Delay Slots

- Compilers put useful instructions in delay slots.
- Branch delay
  - Use instructions from before the branch.
- Load delay
  - Use an instruction that doesn't need the loaded value
  - Or that needs the old value of the register

```
lg:
    addi $sp, $sp, -4
    bne $a0, $zero, big
    sw  $ra, 0($sp)
    j end
    ori $v0, $zero, 0
big:
    jal lg
    srl  $a0, $a0, 1
    addi $v0, $v0, 1
end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
    add $zero, $zero, $zero
```

Branch Delay Slots

Filling Delay Slots

- Compilers put useful instructions in delay slots.
- Branch delay
  - Use instructions from before the branch.
- Load delay
  - Use an instruction that doesn't need the loaded value
  - Or that needs the old value of the register

```
lg:
    addi $sp, $sp, -4
    bne $a0, $zero, big
    sw  $ra, 0($sp)
    j end
    ori $v0, $zero, 0
big:
    jal lg
    srl  $a0, $a0, 1
    addi $v0, $v0, 1
end:
    lw  $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
    add $zero, $zero, $zero
```

Branch Delay Slots

Load Delay Slots

Pseudo Instructions

- Assembly language programming is repetitive
- Some code is not very readable
- The assembler provides some simple shorthand for common operations
- Register $at is reserved for implementing them.

Assembly	Shorthand	Description
or $s1, $zero, $s2	mov $s1, $s2	move
beq $zero, $zero, <label>	b <label>	unconditional branch
Homework?	li $s2, <value>	load 32 bit constant
Homework?	nop	do nothing
Homework?	div d, s1, s2	dst = src1/src2
Homework?	mulou d, s1, s2	dst = low32bits(src1*src2)

Declaring Variables

- Assembler directives declare static variables
  - The reside in the ".data" section
  - Code is in the ".text" section
- Labels allow access
  - Use la (load address)
- More details in B.10 in the text

Example

```
        .data
a_str:
        .ascii "Hello!"
str_len:
        .word 6
        .align 2
some_letter:
        .byte 'l'
        .text
main:
        la  $a0, a_str
...access via $a0...
```

example: count.s

Labels in the Assembler

```
     .text
count:
     la  $t0, foo
     la  $t1, some_letter
     lbu $a0, 0($t1)
     la  $t2, str_len
     lbu $a1, 0($t2)

     beq $a0, $zero, count
     add $zero, $zero, $zero
     bne $a1, $zero, done
     add $zero, $zero, $zero
     addi $t1, $t1, 1
done:
     jal count
     add $zero, $zero, $zero
```

Address	Bytes	Raw Insts.	Asm. Source
		`.text`	
		`count:`	
[00400000]	3c011001	`lui $1, 4097 [some_letter]`	`; 11: la $a0, some_letter`
[00400004]	3424000c	`ori $4, $1, 12 [some_letter]`	
[00400008]	918c0000	`lbu $12, 0($12)`	`; 12: lbu $t4, 0($t4)`
[0040000c]	3c011001	`lui $1, 4097 [str_len]`	`; 13: la $a1, str_len`
[00400010]	34250008	`ori $5, $1, 8 [str_len]`	
[00400014]	91ad0000	`lbu $13, 0($13)`	`; 14: lbu $t5, 0($t5)`
[00400018]	1080fff9	`beq $4, $0, -28 [count-0x00400018]`	
[0040001c]	00000020	`add $0, $0, $0`	`; 17: add $zero, $zero, $zero`
[00400020]	14a00002	`bne $5, $0, 8 [done-0x00400020]`	`; 18: bne $a1, $zero, done`
[00400024]	00000020	`add $0, $0, $0`	`; 19: add $zero, $zero, $zero`
[00400028]	21290001	`addi $9, $9, 1`	`; 20: addi $t1, $t1, 1`
		`done:`	
[0040002c]	0c100000	`jal 0x00400000 [count]`	`; 22: jal count`
[00400030]	00000020	`add $0, $0, $0`	`; 23: add $zero, $zero, $zero`

```
     .data
     .align 2
foo:
     .ascii "Helllo!"
str_len:
     .word 7
     .align 2
some_letter:
     .byte 'l'
```

Address	Bytes	ASCII
[10010000]	6c6c6548 00216f6c 00000007 0000006c	H e l l l o ! l

foo:
0x10010000
= (4097 << 16) | 0

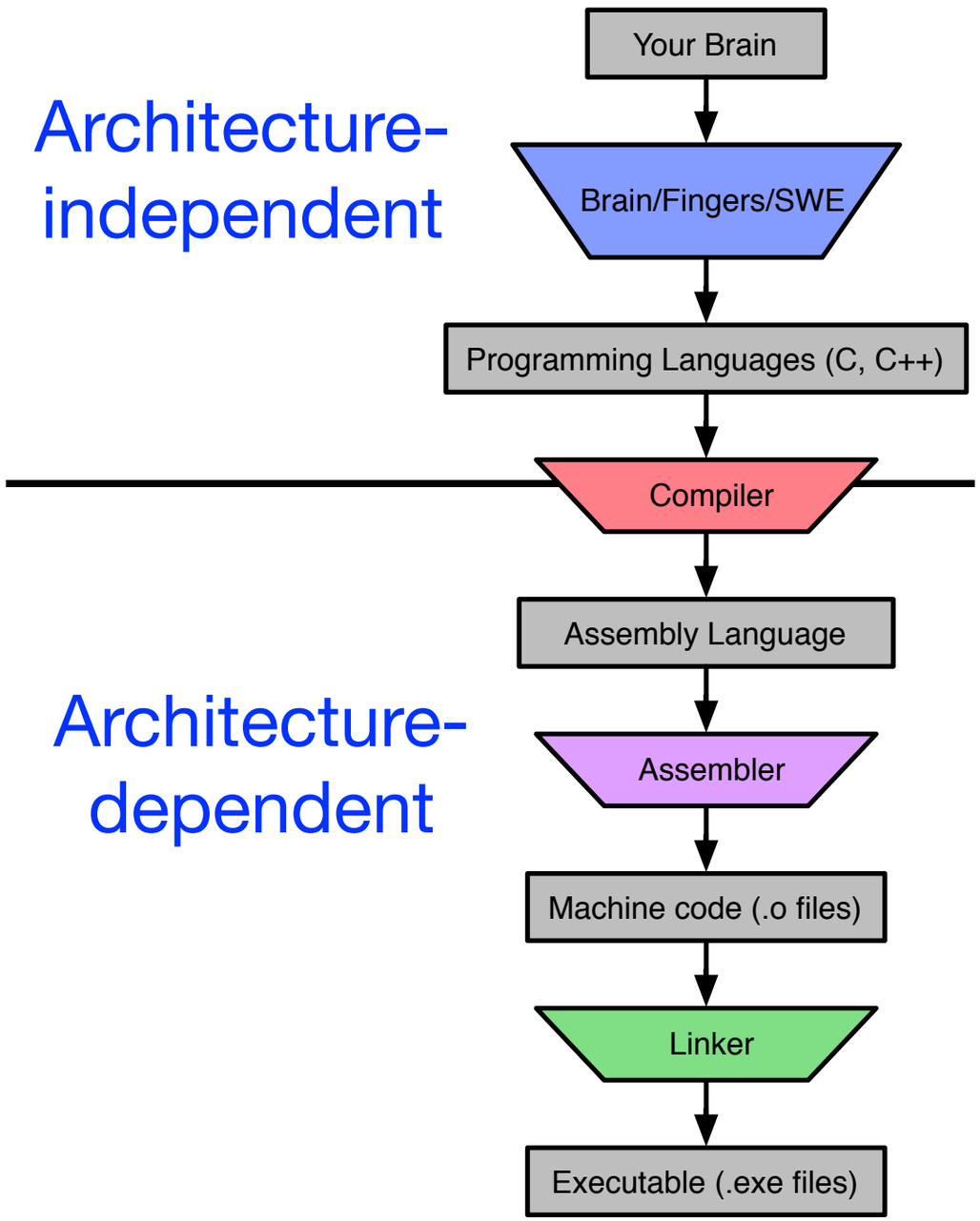
str_len:
0x10010008
= (4097 << 16) | 8

some_letter:
0x1001000c
= (4097 << 16) | 12

From C to MIPS

Compiling: C to bits

Architecture-independent



Architecture-dependent

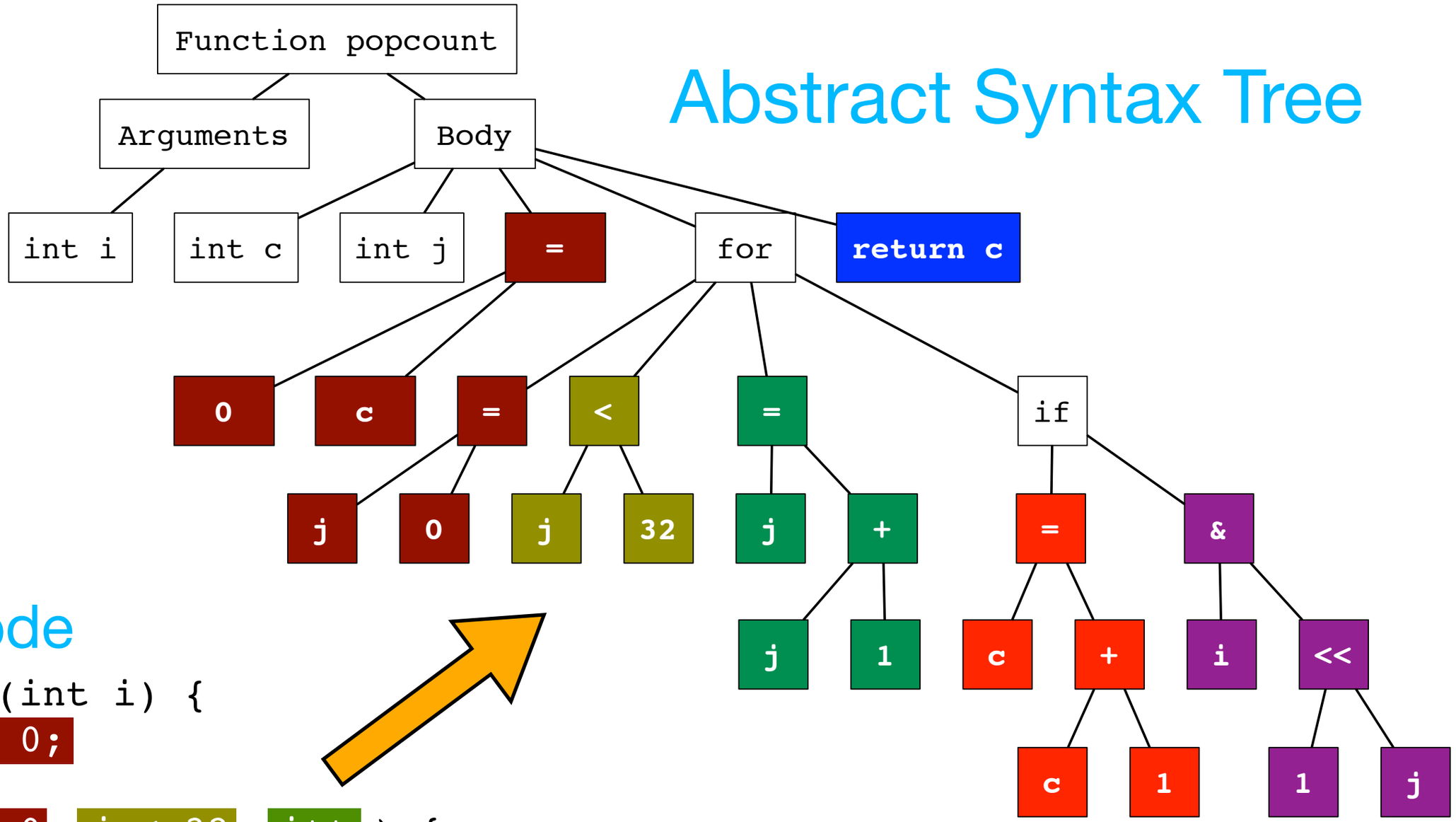
C Code

Count the number of 1's in the binary representation of i

```
int popcount(int i) {
        int c = 0;
        int j;
        for(j = 0; j < 32; j++ ) {
         if (i & (1 << j))
               c++;
        }
        return c;
}
```

In the Compiler

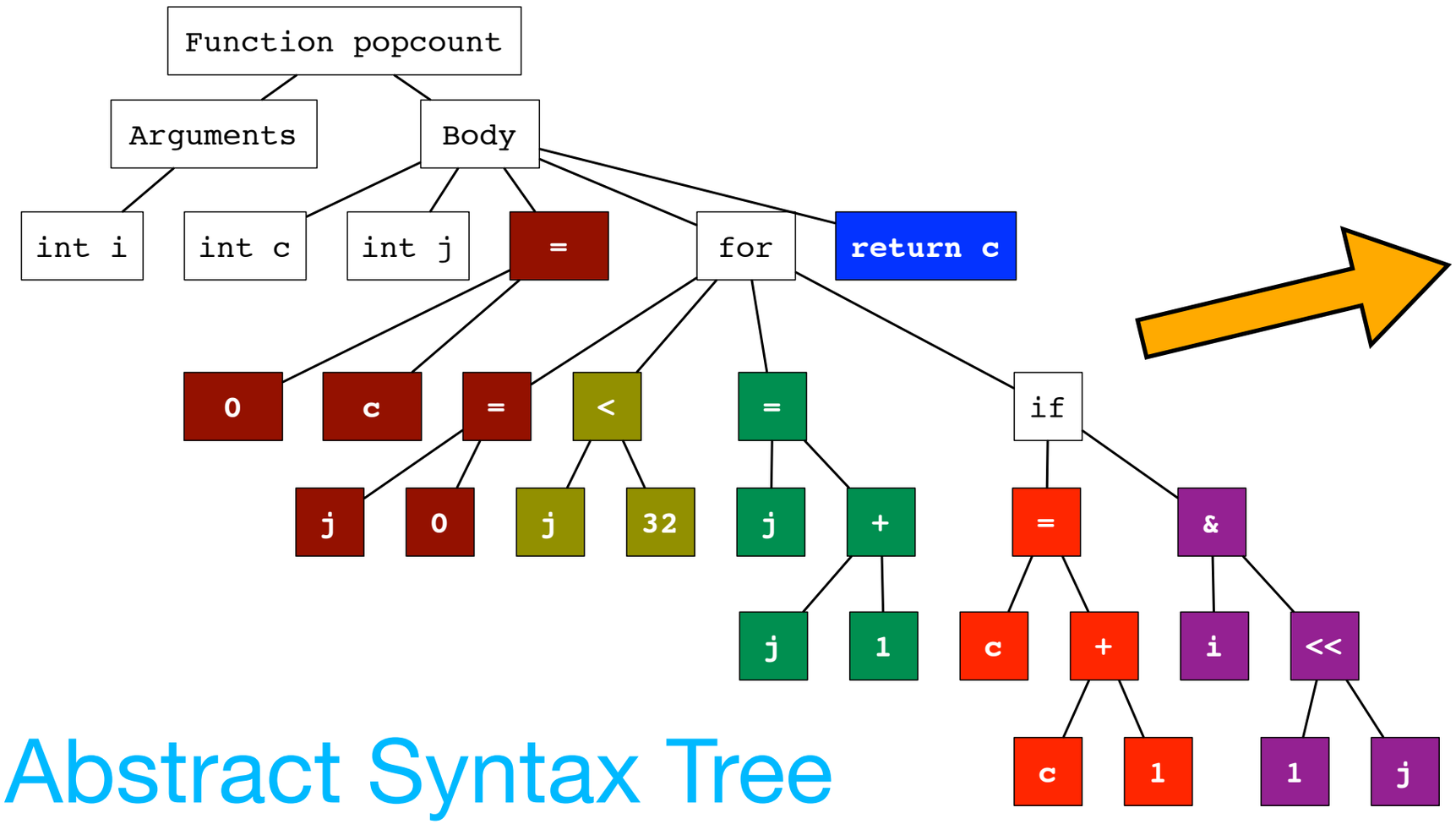
Abstract Syntax Tree



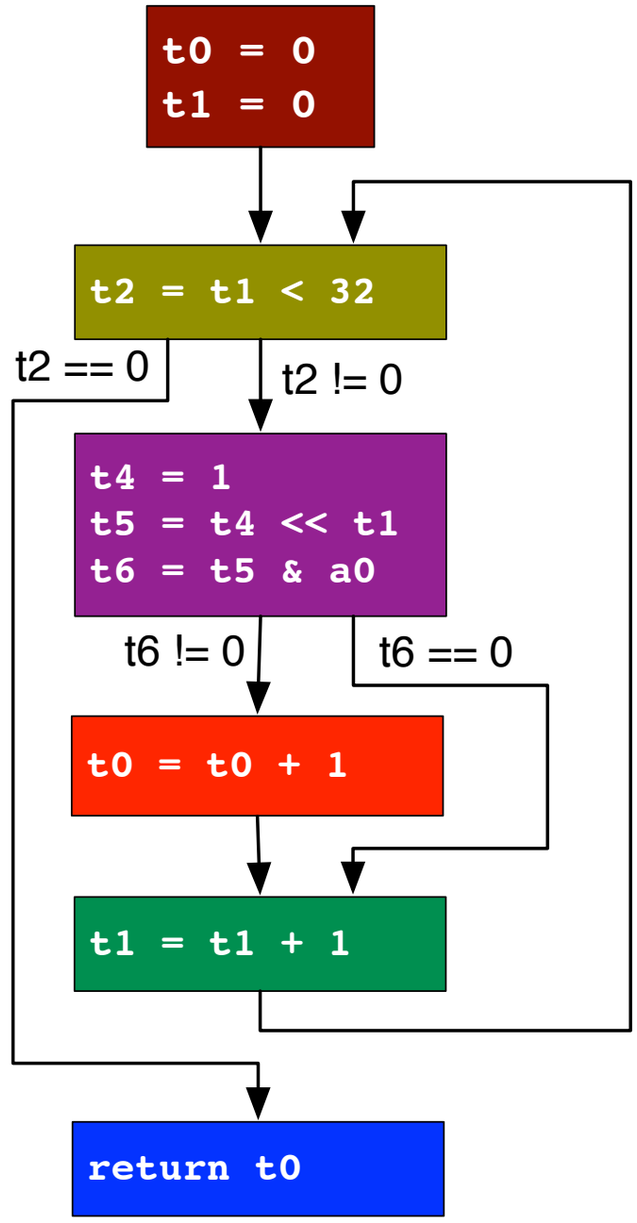
C-Code

```
int popcount(int i) {
      int c = 0;
      int j;
      for(j = 0; j < 32; j++ ) {
       if (i & (1 << j))
            c++;
      }
      return c;
}
```

In the Compiler

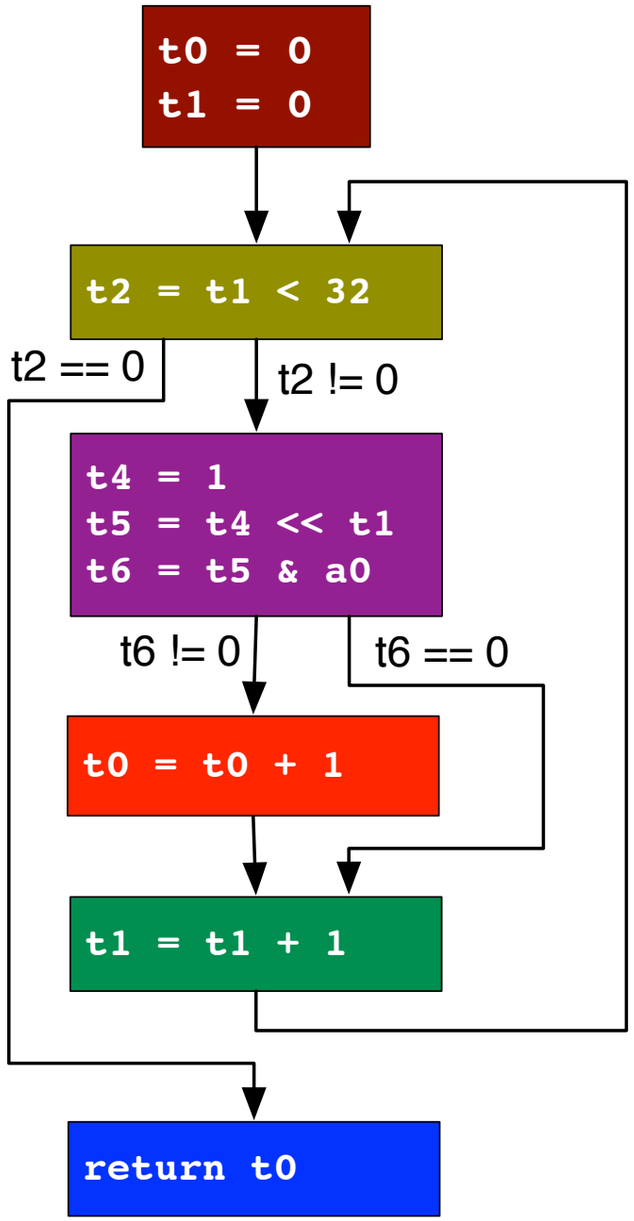


Abstract Syntax Tree



Control Flow Graph

In the Compiler



Control flow graph

```
popcount:
        ori $v0, $zero, 0
        ori $t1, $zero, 0
top:
        slti $t2, $t1, 32
        beq  $t2, $zero, end
        nop
        addi $t3, $zero, 1
        sllv $t3, $t3, $t1
        and $t3, $a0, $t3
        beq $t3, $zero, notone
        nop
        addi $v0, $v0, 1
notone:
        beq $zero, $zero, top
        addi $t1, $t1, 1
end:
        jr $ra
        nop
```

Assembly

In the Assembler

```
popcount:
        ori $v0, $zero, 0
        ori $t1, $zero, 0
top:
        slti $t2, $t1, 32
        beq  $t2, $zero, end
        nop
        addi $t3, $zero, 1
        sllv $t3, $t3, $t1
        and $t3, $a0, $t3
        beq $t3, $zero, notone
        nop
        addi $v0, $v0, 1
notone:
        beq $zero, $zero, top
        addi $t1, $t1, 1
end:
        jr $ra
        nop
```

```
00110100000000010000000000000000
00110100000001001000000000000000
00101001001010100000000000100000
00010001010000000000000000001001
00000000000000000000000000000000
00100000000010110000000000000001
00000001001010110101100000000100
00000000100010110101100000100100
00010001011000000000000000000010
00000000000000000000000000000000
00100000010000100000000000000001
00010000000000001111111111110110
00100001001010010000000000000001
00000011111000000000000000001000
00000000000000000000000000000000
```

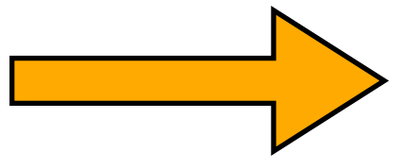
Assembly

Executable Binary

In the Compiler

```
int popcount(int i) {
    int c = 0;
    int j;
    for(j = 0; j < 32; j++ ) {
        if (i & (1 << j))
            c++;
    }
    return c;
}
```

C-Code



```
popcount:
        ori $v0, $zero, 0
        ori $t1, $zero, 0
top:
        slti $t2, $t1, 32
        beq  $t2, $zero, end
        nop
        addi $t3, $zero, 1
        sllv $t3, $t3, $t1
        and $t3, $a0, $t3
        beq $t3, $zero, notone
        nop
        addi $v0, $v0, 1
notone:
        beq $zero, $zero, top
        addi $t1, $t1, 1
end:
        jr $ra
        nop
```

Assembly

Top 5 Reasons to Use Assembly

1. You are writing a compiler, so you have no choice.
2. You want to understand what the machine is actually doing (e.g., why your code is slow). In this case, you just need to read assembly.
3. You need to do things that are not possible in C
   - e.g., It is not possible to implement locks correctly in C.
   - e.g., Many other low-level OS operations can't be expressed in C.
4. It's faster sometimes
   - Compilers mechanically convert C to assembly, and they may not emit the fastest code possible.
   - You might know better...
     - The compiler might not recognize opportunities to apply specialized instructions (e.g., SSE vector instructions)
     - You might be desperate for performance, and be able to squeeze a bit out here or there.
   - But probably not.
     - Modern compilers are very good.
     - Unless you know exactly why you want to use assembly, you shouldn't.
     - Even then, you should try to find a way to do it in C (e.g., Compiler "intrinsics" to force the compiler to emit SSE instructions, or restructuring your C code)
5. You are doing cse141 homework